PAWAN - A MACH BASED UNIX SYSTEM – I

By

ASHISH SINGHAI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1992

PAWAN - A MACH BASED UNIX SYSTEM

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF TECHNOLOGY

by

Ashish Singhai

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

April, 1992

CERTIFICATE

It is certified that the work contained in this thesis entitled "PAWAN -- A MACH BASED UNIX SYSTEM (I)" by *"Ashish Singhai"* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Barua

(Dr. Gautam Barua)
Assistant Professor
Dept. of CSE
IIT Kanpur

April 1992

Acknowledgements

I wish to thank my supervisor, Dr. Gautam Barua for his guidance and help throughout the course of thesis. Thanks are also due to Dr. Somenath Biswas, who never let me feel I was away from home. I am also grateful to T. Srikanth, who explained to me the virtues of life and those of unix.

Other members of the IITK-MACH group, Gopal, Mrinal and Pull were always a great company. I am happy and we could realize the ideas we planned together. Deepak Gupta deserves special thanks for helping in signal implementation. He was always ready to hack anything and also lifted spirits whenever they went low. Also, Shreesh Jadhav was a wonderful company and made me realize there is more to life than I thought.

Staying with Tarun, Ajay, Sheenoo, Telya, Kiran, Rajiv and Amit was wonderful, who were always planning some junket. Swarup provided the essential component of life --- humor.

I also wish to express my gratitude to D2, who never understood but always had more faith in me than I had in myself.

Software engineers and lab staff at CSE Lab were very helpful. I wish to thank them for their help. User community of CSE Lab is also worthy of thanks --- for their patience and endurance while we experimented with HCL.

ABSTRACT

In this thesis the design and implementation of program execution facility, the exec server and the signal server for PAWAN are described. PAWAN is an operating system based on MACH3.0 micro kernel. It consists of a set of user state servers which run on the MACH micro kernel and provide a UNIX like user environment.

As the first step in PAWAN's development, the source code for the i386 version of MACH3.0 was obtained from Carneige Mellon University which was then ported onto the MC68020 based HCL Horizon III machine in the CSE department. Various services like the file service, naming service, signals, program execution, terminal handling and networking were added to enhance the micro kernel. Keeping in mind the large user group and the huge number of utilities available for UNIX, PAWAN's programming environment was designed so that source code of UNIX programs could be used with minimal modifications. UNIX semantics are redefined to suit a server based implementation while preserving functionality. PAWAN is presently running on Horizon III machines.

Sophisticated user environments can be provided by porting existing utilities on PAWAN with minimal effort. Advanced features like network-wide shared memory and load balancing can also be built on the foundation laid by PAWAN. It is expected that PAWAN will provide the required platform for distributed systems research at IIT Kanpur.

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

The size and complexity of UNIX have increased tremendously over the last few years. With the advent of distributed computing and multiprocessors, the need to incorporate new features in it has constantly been felt. This has reduced the advantages of simplicity and modifiability -- the hallmarks of UNIX during the 1980s. Moreover, there was a clear need to allow the underlying system to be transparently extended to allow user-state processes to provide services which in the past could only be integrated into UNIX by adding code to the OS kernel. These factors led to the development of a new generation of distributed operating systems, for example, Accent, Locus, MACH, Amoeba, V-system and Sun NFS-based products. MACH (developed at Carnegie Mellon University) is one such operating system which tries to 'kernelize' the functionality of UNIX by providing a small set of primitive functions that allow more complex services to be built as user level servers.

This thesis describes the design and implementation of PAWAN, a MACH based UNIX System, which provides an environment for the development of distributed operating systems. PAWAN runs on a network of MC68020-based uniprocessors at IIT Kanpur. It provides traditional UNIX services like file, network, signal and terminal handling through separate user-state servers. This is a significant and novel departure from the single UNIX Server implemented over MACH[Tev87a].

1.1 MACH Operating System

MACH is a multiprocessor operating system developed at Carnegie Mellon University[Acc86a]. The design of MACH was influenced by experience gained with a previous system developed at CMU called Accent. Many MACH features are derived from it. The approach in both Accent and MACH has been to design and build a kernel that is suitable for distributed systems and is also able to emulate UNIX. This is quite different from the approach in Locus[Wal83a] where UNIX was extended to work in a distributed manner with the resulting system having some of the same limitations as UNIX. MACH is a light weight kernel running in each computer with services such as file system, network service and process management outside the kernel. These services replace the system calls found in conventional operating systems such as UNIX. The model provided by MACH is a service model in which objects are managed by servers, and clients make requests for operations on objects by using remote procedure calls.

MACH has been designed to run in diverse hardware configurations, such as, uniprocessors, tightly-coupled shared memory multiprocessors, loosely-coupled muliprocessor with limited, or differential access to shared memeory. It has been implemented for Vax, MicroVax, IBM PC/RT, Perq, Encore, Sequent and Sun machines.

To effectively utilize such machines, MACH provides the following features :

- Separation of typical process abstraction into a task and thread.
- Powerful virtual memory primitives, allowing sharing via inheritance mechanism with copy-on-write implementation.
- A communication mechanism that is transparently extendible over a network.

1.2 Survey of Other Distributed Systems

1.2.1 Locus

Locus is a distributed version of UNIX which provides high degree of transparency of file location and some degree of transparency of location of execution. The features of Locus are system-wide file naming, a file storage system with replication and the ability to run processes remotely. Locus appears to clients like one giant UNIX system, with all of the computers playing both server and client roles and with UNIX file access, process creation and interprocess communication primitives implemented transparently across the network. Relation between the kernel and a user is the same as in UNIX except that the local kernel may route it to another kernel to execute it by using a kernel-to-kernel remote procedure call.

1.2.2 Sun NFS

Sun's NFS is a distributed file system over which many distributed services like Yellow Pages are built. NFS provides remote access to conventional UNIX file stores. In NFS, it is possible to mount remote file directories that have been exported by other computers as part of the file name space in local store. Once the appropriate remote file stores have been mounted users and client programs need not be aware of the location of the files. A user can use the standard UNIX primitives, so programs written to operate on local files can be used with remote files without modification.

The NFS service is implemented in terms of remote procedure calls between kernels. All kernels are both clients and servers of files and directories. The NFS software consists of a set of extensions to the UNIX kernel and a set of library procedures to enable user-level programs to mount remote files and to export local file systems. The kernel extensions enable a UNIX kernel to act as a client to other kernels when accessing remote files and to act as a server of local files when receiving access requests from other kernels.

1.2.3 Amoeba

Amoeba is a distributed operating system[Tan90a] which runs on a local network of work stations, pool of processors and specialized servers connected to other LANs through a gateway. Amoeba was implemented with many of the components of conventional OS, such as the file service, outside the kernel. The kernel includes facilities for creating processes as clusters of threads, and inter-process communication based on a triple of message passing primitives designed to support RPCs -

* Request, for use by clients to make remote calls.

* GetRequest and PutReply for servers to receive and respond to server calls.

A user views Amoeba as a collection of processes and information objects maintained by servers. The objects and servers are identified by sparse capabilities. A capability allows a processor to perform certain operations on the object it names.

Several different file systems have been built on Amoeba, including the UNIX file system with UNIX calls, a flat file service and an advanced transaction-based file service called Free University Storage system (FUSS).

1.2.4 V kernel

The V kernel is a distributed operating system designed for a cluster of computer workstations connected by a high performance network. The design of V kernel relies on the assumption that high performance communication is the most critical factor for distributed systems and protocols[Che88a]. In the V distributed system, separate copies of the kernel run on different machines which cooperate to provide a system abstraction of processes in address space, communicating using a set of communication primitives. The V kernel appears to the applications as a set of procedural interfaces that provide access to the system services.

The V Kernel provides a network-transparent abstraction of address spaces, lightweight processes and fast interprocess communication. Multicasting, a naming protocol and a uniform I/O interface are also provided in the kernel. These facilities together provide a basic framework for implementing variety of services like pipe server, internet server, file server, etc.

1.3 Motivation

At IIT Kanpur, a UNIX compatible OS "IITKIX" was developed over the past few years to experiment with various operating system concepts[Das89a]. However, with the advent of distributed computing environments consisting of networks of uniprocessors as well as multiprocessors, we felt the need for a new distributed OS to act as a platform for future research on process migration, distributed shared memory, load balancing and so on. As mentioned earlier, MACH has been designed with the intent to integrate both distributed and multiprocessor functionality. It therefore turned out to be the ideal choice for our purpose.

We obtained the source code of MACH3.0 Micro Kernel for i386 based-systems. Since the hardware description for i386 systems was not available, we decided to port it onto an MC68020-based mini, the HORIZON III. The next step was to build various distributed services in PAWAN as MACH3.0 does not provide a file system, tty i/o, network support or other UNIX features like signals and processes (fork, exec, etc).

We decided to build a UNIX-like environment with source code compatibility since the user community is familiar with UNIX and the existing code for UNIX systems could be reused. Binary compatibility with UNIX was not an issue since it would lead to inefficiency and lack of flexibility.

We decided to implement the following servers : Exec Server, File Server, File Pager, Name Server, Network Server, Signal Server and Terminal Server.

This exercise has helped us to enhance our knowledge about UNIX and MACH internals, operating system porting issues and the design and implementation of distributed services.

1.4 Outline of Rest of the Thesis

Introduction to MACH including its underlying philosophy, abstractions and functions has been dealt with in chapter 2. The overall design of PAWAN is described in Chapter 3, along with its server environment. Chapter 5 discusses issues in porting the MACH3.0 kernel. Chapter 5 describes the signal server which provides UNIX style signal facilities. Next chapter is on program execution facilities and ES, the exec server. The thesis concludes with Chapter 7 with the possible future extensions to PAWAN.

PAWAN has been developed in a group at IIT Kanpur. Details of the servers not included in this report can be found in the theses of the other members of this group[Gop92a,Rao92a,Bar92a].

CHAPTER 2

MACH OVERVIEW

This chapter presents a brief overview of the MACH kernel. The underlying philosophy, kernel abstractions and features of MACH are examined. More details on MACH can be found in the references[Acc86a,Tev87b,You87a].

2.1 MACH Philosophy

MACH has been designed with a goal of creating integrated computing environments, consisting of networks of uniprocessors and multiprocessors[Tev87a]. The basic functionality of the kernel is designed to support the integrated computing environment of the future.

1. MACH supports diverse architectures (UMA, NUMA, NORMA).

2. It can handle range of communication speeds (LAN, WAN, tightly-coupled multiprocessor).

3. MACH is a new OS organization with

   * small number of abstractions.

   * kernel/OS server model.

   * network transparent and object oriented.

   * integrated memory and communication.

The central component of a MACH-based operating system environment is the MACH kernel. Typical operating system services are lavered above the MACH kernel as a set of system servers. Since the MACH kernel is a simple base for system servers, it is readily portable and adaptable to the wide range of computing architectures present today and anticipated in the future. Since servers provide most of the traditional system services, different software environments can be run easily in different hardware environments.

2.2 Basic MACH kernel functionality

MACH can be viewed as being split into two components. The first is the small, extensible system kernel which provides scheduling, virtual memory and interprocess communications, and the second component is the several, possibly parallel, operating system support environments, which provide emulation for established operating system environments such as UNIX.

The MACH kernel supports five basic abstractions: Task, Thread, Port, Message and Memory Object.

* A task is an execution environment and is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources (such as processors, port capabilities and virtual memory).

* A thread is the basic unit of execution. It consists of a processor state necessary for independent execution. A thread executes in the virtual memory and port-rights-context of a single task.

- A port is a simplex communication channel, implemented as a message queue managed and protected by the kernel. A port is also the basic object reference mechanism in MACH. Ports are used to refer to objects; operations on objects are requested by sending messages to the ports which represent them.

- A message is a typed collection of data objects used in communication between threads. Messages may be of any size and may contain inline data, pointers to data, and capabilities for ports.

- A memory object is a secondary storage object that is mapped into a task's virtual memory. Memory objects are commonly files managed by a file pager, but as far as the MACH kernel is concerned, a memory object may be implemented by any object (i.e. port) that can handle requests to read and write data.

Message-passing is the primary means of communication both among tasks, and between tasks and the operating system kernel itself.

2.3 MACH Features

The MACH kernel functions can be divided into the following categories:

- task and thread creation and management facilities,

- * virtual memory management functions,
- * basic port and message primitives,
- * operations on memory objects.

2.3.1 Tasks and threads

MACH divides the typical UNIX process abstraction into two orthogonal abstractions: the task and thread[Tev87c]. MACH allows multiple threads to exist(execute) within a single task. On tightly coupled shared memory multiprocessors, multiple threads within the same task may execute in parallel. The context switching time for threads of the same task is small. Operations on tasks and threads are invoked by sending a message to the task kernel port and thread kernel port. Threads may be created, destroyed, suspended and resumed. The suspend and resume operations, when applied to a task, affect all threads within that task. In addition, tasks may be created and destroyed. A standard UNIX fork operation takes the form of a task with one thread creating a child task with a single thread of control and all memory shared copy-on-write.

2.3.2 MACH Virtual Memory (VM) Management

MACH has implemented a new, portable memory management system whose main features are:

* Architecture independence: support for a wide range of paged architectures[Tev87b].

* Distributed system and multiprocessor support: features such as shared memory and integrated memory management and message passing[Ras87a].

* Advanced functionality: especially, copy-on-write, shared libraries, memory-mapped files and user-implementable memory-objects[You87a].

MACH Virtual memory interface is divided into several functional groups:

* Address space manipulation including allocation and deallocation of virtual memory of a task at a page level.

* Memory protection allowing flexible use of different memory protection hardware.

* An inheritance mechanism for creation of address spaces in tasks.

* Miscellaneous primitives that formalize access to statistics maintained by the MACH kernel, access other task's virtual memory and describe a task's address space.

The new virtual memory design provides :

* Large, sparse address spaces (needed for large-scale AI application like speech, vision etc)

* Memory mapped files and user-provided storage objects (memory objects)[Tev87d].

- Read/write and copy-on-write sharing (makes possible transparent parallel programming and networking).

- Integrated virtual memory and communication:

  - Large messages sent without physical copies being made (database management, graphics and AI).

  - Memory object capabilities can be passed in messages.

  - Network shared memory with variable consistency constraints.

An important feature of MACH's VM is the ability to handle page faults and pageout data requests outside the kernel[You87a]. When VM is created, special paging tasks may be specified to handle paging requests. MACH also provides some basic paging services inside the kernel through a default pager task. Memory allocated with no pager specified is automatically zero-filled and its pageout/pagein is handled by the default pager.

2.3.3 Interprocess Communication

The MACH interprocess-communication facility is defined in terms of ports and messages and provides both location independence, security and data type tagging. Ports are used by tasks to represent services or data structures. Access to a port is granted by receiving a message containing a port capability. Port capabilities include:

- send rights, which correspond to the capability to send a message to a port. Send rights may be held by any number of tasks.

- receive rights, which correspond to the capability to receive a message on a port. receive rights may be held by only one task.

The MACH kernel automatically queues messages for tasks executing on its machine. However, transmission of messages between separate MACH kernel hosts should be performed transparently by an intermediate server task, known as Network Message Server.

2.4 The I/O structure in MACH

MACH employs an I/O structure substantially different from that of existing systems. Peripheral devices are accessed through the device server port. Device server is implemented as a kernel object. The interface to the device server is through IPC. This section discusses the device server, its services and special features.

2.4.1 Device Switch

Device switch dev_name_list describes the devices potentially attached to the system. Each entry in this table describes the entry points to the driver. Another table Devs gives the bus specific description of devices. Noticeable differences from UNIX are:

* Integration of character and block devices in the same table.

* Kernel just houses the drivers and does initialization. It does not do any other operations on devices.

2.4.2 Normal Input and Output

Device server allocates a port for a device in response to device_open() request. This is the port which is then used to perform I/O on that device. Such ports can then be handed out to various tasks on the discretion of some trusted agent. It may be noted here that send rights to device server port imply complete control over all the devices, whereas access to the port corresponding to a device imply control over that particular device.

Normal I/O is done using MACH IPC on the device port. The I/O can be inband or out of band as suitable for specific devices, inband message passing being more suitable for small amounts of data. Out of band device I/O benefits from copy-on-write scheme used for message passing. It obviates memory to memory copying of huge amounts of data, thus improving efficiency.

Data are not buffered by device server to avoid hardwiring the policies within it. The applications which use its services are expected to employ their own buffering schemes.

2.4.3 Asynchronous Input

Asynchronous input is handled in a novel way in MACH through a special entry point in the device switch. This entry point device_set_filter(), associates a filter (a boolean function) with a port. The input from device is then filtered and queued at the specified port if filter output is TRUE. This scheme makes it possible to have user level implementations of services which traditionally resided in kernel e.g. network service.

This interface was used in an earlier design of the network server which had a user level implementation. The TTY server uses this interface for filtering special keyboard characters. For example, break key in cbreak mode is provided to the server in the control stream rather than the data stream -- the filter recognizes it and acts in differently.

2.4.4 Iodone processing

Functions performed at the completion of an I/O request are encapsulated in iodone(). In UNIX they generally consist of waking up the process waiting for it and are short enough to be executed from within the interrupt handler. MACH can not afford performing iodone processing in the interrupt routine since it involves data transfer too. Hence iodone_thread is used to perform these functions.

CHAPTER 3
DESIGN OF PAWAN

This chapter describes the design goals of PAWAN, the user state servers in it, and some general issues related to security, user identification, user interface, concurrency and user-state maintenance in servers. The system initialization procedure is mentioned at the end.

3.1 Design Goals

PAWAN uses several user state servers to provide UNIX services in a completely transparent manner. The motivating factor behind this design was our development environment. In a shared memory multiprocessor machine, a single UNIX server implemented through system call and exception redirection (emulation) might prove to be more efficient ([Acc86a,Tev87d,You87a]) than our design. However, in a loosely coupled LAN based environment such as ours, a centralized UNIX server which services every user-request, system call or exception in the network will obviously be extremely inefficient. Also, system call and exception redirection by itself is prohibitively expensive on a LAN due to the communication overheads involved. Moreover, MACH itself was envisaged to reduce the number of system calls and hence the number of fundamental concepts for a user to deal with. The only other alternative without using multiple servers is to implement a single distributed UNIX server consisting of several separate UNIX servers running on different machines and communicating with each other transparently (e.g. LOCUS, section 1.2.1). Experiences with LOCUS[Wal83a] have demonstrated the sheer magnitude of this task and the extreme difficulty in upgrading a huge monolithic system like UNIX.

Another important goal was to make use of existing UNIX code wherever possible (e.g. file and network servers) so that greater effort could be spent in improving the efficiency and modularity of our system. UNIX source code compatibility has also been a major design goal in the relevant servers since users are used to the UNIX programming environment and existing UNIX programs could be run without major modifications. However, binary UNIX compatibility was not the driving factor as it would again imply system call and exception redirection which is unsuitable in a loosely coupled system.

We therefore decided to split UNIX services into several independent parts, each of which could be handled by a separate server with an accompanying library. We came up with six user-state servers, the Environment, Exec, File, Network, Signal and TTY servers which are absolutely essential to PAWAN.

1. The Environment Server

It handles naming in our distributed system by providing a mechanism for tasks to share named variables (ports, strings and environments), an environment being a set of variables. It also provides public and private ports facility to well known servers and is central to system initialization. It is an extension of the MACH Environment Manager[Tho88a] and has no UNIX equivalent.

2. Program Execution

   Execve() is implemented as a library routine which facilitates execution of object modules with the help of a resident piece of code which is inherited in a new task from its parent and runs as a co-routine with the main program. The Exec Server intervenes in the execution of setid programs since authentication update is involved.

3. The File Server

   It is a 4.3BSD compatible MACH based file server. It uses 4.3BSD file system code and provides both UNIX style file I/O and a pager based interface with external memory management. It can easily be integrated with the network server to provide transparent network-wide file access and shared memory.

4. The Network Server

   It provides a socket based connection oriented and datagram services using the IP suit of protocols. The socket interface uses TCP code from BSD4.3. It is implemented as a privileged kernel task with user interface through system calls.

1. All servers have a well known (public) port to receive user requests and a secure (private) port to receive privileged requests from other servers. All public ports are registered in the Env Server, the Env server's public port itself being available to all tasks by default. Users first query the Env server and obtain the public port of the server they wish to contact, and then send their request to it (figure 3.1).

2. When a server receives the first request from a user task on its public port, it returns a port to the user task for communication with it. From then onwards, all future requests to the server can be made on this port. Servers have threads waiting on the ports returned above so that multiple requests can be served simultaneously. This schematic is useful with servers which maintain state about their clients over long periods of time.

3. Servers use their private ports for secure communication among themselves (e.g., changing the privileges of a user task, etc.). These ports are not available to normal users.

4. All requests are handled as Remote Procedure Calls (RPCs) and all servers use the MACH Interface Generator (MIG)[Dra89a] in their implementation of the user and server stubs.

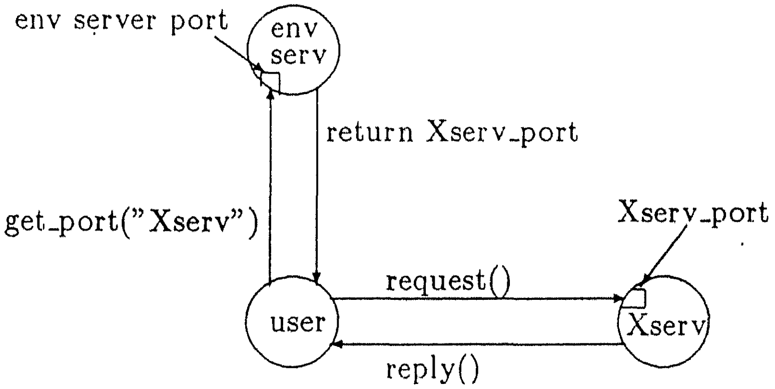

Figure 3.1: Interaction between a User and a Server

3.2.2 Synchronization

In a multi-threaded user level server environment, synchronization is of utmost importance. In UNIX where all services are implemented through system calls, synchronization could be achieved by simply raising the processor priority level appropriately to block interrupts, since a system call could never be interrupted anyway. At the user level, things are quite different. Explicit synchronization through locking variables must be done before modifying shared data structures. This might also involve the use of sleep and wakeup synchronization. The MACH Cthreads Package[Coo88a] provides all these primitives at the USER level and has proved very helpful in this context -- most of the servers in PAWAN use it in their implementation.

3.2.3 Unique Identification

UNIX identifies each process by its pid, which is globally unique. In MACH, access control is based on a per-task basis since the task is the unit of resource allocation. To enforce UNIX-like authentication at any server, we need a per-task-id guaranteed to be unused for a significant period of time after its termination, just as the UNIX pid is. This tid has to be sent in all requests to enable servers to identify and authenticate the user task.

There are two solutions to this problem:

1. Each server uses the send rights of the kernel port of a user task (guaranteed to be unique as long as the server exists) to identify it. The problem with this approach is that each server has a different concept of what the tid of a user-task is (since rights of a port are not unique to all tasks). But as long as rights are transferred between tasks, they get appropriately translated in the new task (as explained in section 3.2.5). Some servers (e.g., Signal Server) which provide explicit UNIX services also maintain an integer pid corresponding to every user task since in some cases as the tid is either not enough or unsuitable to identify it.

2. To have a tid-server inside the kernel which automatically generates a globally unique tid for every task created (just like UNIX). This solution is avoided because it unnecessarily increases the complexity of the kernel which is against the basic MACH philosophy.

3.2.4 User States in Servers

Servers in PAWAN are stateful, i.e. some state is maintained for every task accessing the server, tasks being identified by their tids or pids or other mechanisms. This state can be maintained in:

User space

In this case, the servers have to trust user information and no protection can be guaranteed. Also, every request message has to carry all information about the current user state in it and the corresponding reply has to update it. Though servers can be stateless in this case, the overheads incurred are too high, and this choice was discarded.

Kernel space

In which case, either the kernel has to be modified, or the servers have to be inside the kernel -- both alternatives being against our fundamental design goals.

Server space

In which case, the user is identified through a tid, and servers become stateful. This alternative does not compromise security, helps in developing modular user state servers and is also efficient since it reduces the number of IPC messages. That is why it has been adopted in PAWAN.

The per-task state contained in each server is:

1. The authentication information if any (e.g. uids, gids, groups etc.).

2. The server state corresponding to a task (e.g., ports being used for communication with user tasks and some private data-structures opaque to the user).

All servers in PAWAN have been implemented assuming that they know about the death of the user task in some fashion. This information can come from:

1. A notify message from the kernel if the servers request a notification on the death of the user task identified by its tid (send rights of its kernel port).

2. The exit library routine itself (on a normal termination) or in some other way on task termination.

3. The servers can also examine the kernel ports of the user tasks for which states are maintained to determine if the tasks are dead, and if so, do whatever cleanup is required (e.g. deallocate communication ports, free memory, etc.). This can be triggered periodically, or whenever there is a resource shortage.

Other details about state maintenance are described below.

3.2.5 Credentials and Authentication

Every server implementing UNIX style authentication based on uids and gids needs to have a reliable way to get them for every task in the system (assuming every server has root privileges). The credentials of a task can change implicitly on exec (setuid and setgid) or by an explicit change, as on login.

Logical solution seems to have a separate Authentication server which stores a mapping from the tid (or pid) of a task to its UNIX authentication information. Any server which wants to authenticate a request does so by asking the Auth Server. Updates to credential information can be done only by privileged servers -- generally the Exec, File and Network Servers. Normal users need to send only their tids (or pids) to servers. Servers authenticate them by the help of Auth Server.

This approach is very elegant and has been used in many distributed systems ([Bir84a,Che88a,Tan90a]). The advantages are obvious -- authentication process is logically separate and hence various forms of protection can be implemented, e.g. capabilities (login-id and password tuples) or Access Control Lists -- currently not supported in UNIX. Also, servers need to maintain the state of only those tasks which access them and need not be aware of every task in the system; state needs to be made only when the user task sends the first request to the server. We have left authentication to future extensions since it is not central to our goals.

3.3 System Initialization

The first user program executed by the MACH kernel after kernel startup is the Environment Server. It creates each server task, initializes its environment and registers the public and private ports of each server in the environments of all servers. Each server task is then directly executed (without going through the Exec server) by making use of a read only file-system (bootload) implemented inside the MACH kernel. As each server comes up, it initializes the state corresponding to each well known server in the system (identified by the send rights of their private ports). Then, each server waits for user requests on its public port and privileged operations on its private port. Initialization is complete when all servers are ready to receive requests. Users can start interacting with PAWAN when the TTY server gives a prompt and is ready to accept user input.

The overall view of PAWAN and its server environment, and various important issues in its design and implementation have been discussed in this chapter. This general background to help the reader appreciate the later chapters dealing with individual servers better.

CHAPTER 4

PORTING OF MACH KERNEL

4.1 Introduction

This chapter describes the issues involved in porting MACH3.0 from the original i386 version onto an MC68020-based system, the HORIZON III. This machine was chosen for the following reasons:

1. The hardware description and device characteristics of the i386 machines in our department are not available due to proprietary reasons. Even though CPU-dependent code could run without any modifications as such, it is not possible to run MACH code as a whole without porting the device drivers.

2. However, we had access to three Horizon machines interconnected by the department ethernet. The hardware manuals for the devices connected to it, device driver sources for 4.2 BSD UNIX and the source code for the IITKIX operating system developed for these machines were also available.

Since the Hardware Characteristics of the Source and Target machines are quite different, a lot of time and effort was required to port the MACH micro kernel. In the first step, the machine-dependent portions of the code were identified and their semantics thoroughly understood. They were then rewritten in C and MC68020 assembly language. Most of the time was spent in debugging the new kernel and "fitting in" the changes made in a modular and consistent manner. This task became more difficult due to the absence of a kernel debugger -- the only debugging-aids available were two routines: siogetchar() and sioputchar() obtained from IITKIX code. These functions access an RS232C serial port of the CPU card connected to the Console and print/wait for a character from it synchronously. Break points had to be inserted manually using these functions and every piece of code had to be traced to ascertain its correctness.

4.2 Target machine description

The HCL Horizon system ([HCL1], [HCL2], [HCL3]) runs BSD4.2 UNIX. It is based on the MC68020 microprocessor with the following peripherals:

* 16 RS232C serial i/o ports for terminals.

* 2 100MB disks.

* 4MB main memory.

* 1 cartridge drive.

* 1 parallel printer port supporting Centronics printers.

The MC68020 CPU [MCK20] has eight 32-bit data registers, seven 32-bit address registers, two 32-bit stack pointer registers, a 32-bit program counter (PC), and a 16-bit status register. The status register contains five status flags, three interrupt mask bits, one bit to set either supervisor or user mode, two bits to set trace mode and a bit to indicate that interrupt stack (rather than the supervisor stack) be used for interrupt processing. Memory mapped I/O is used for accessing the peripheral device registers. The CPU can operate in either of two modes -- user mode or supervisor mode (protected or kernel mode).

4.3 Areas of Modification

The following major areas of modification of MACH machine-dependent code were identified:

1. CPU dependent code.
   - * Initialization code specific to the CPU.
   - * Interrupt and exception handling code.
   - * Thread and task context switching.

2. Virtual memory management code.
   - * The Pmap module and MMU driver.
   - * VM fault handling and recovery.
   - * Page table consistency on context switch.
   - * Kernel startup and VM initialization.

3. Device Drivers.
   - * Terminal driver.
   - * Disk driver.
   - * Network driver.

4.4 Porting Kernel Locore

4.4.1 Exception Handling (Traps and Interrupts)

The MC68020 provides extensive exception processing logic including a very complete set of external interrupts as well as internally initiated exceptions upon detection of various faults, traps, and so on. The internally detected errors are addressing errors, privilege violations, illegal and unimplemented opcodes, instruction traps (trace etc). The externally generated exceptions are bus errors, reset and interrupt request.

Exception Vector Table is central to the MC68020 exception processing sequence. It occupies 1024 bytes of memory, from physical addresses 0x000000 through 0x0003ff. The table is organized as a 256 four-byte vectors. Each vector is a 32-bit address which will be loaded into the PC as part of the exception processing sequence.

Trap Handling

The generic exception handler routine does the following :

1. Save the registers.

2. Determine the interrupt number from the exception frame pushed onto the stack.

3. Call the trap routine to process the individual traps.

4. Restore registers.

5. Return from exception to the original program or abort.

One of the exceptions that needs special servicing is the bus error. A bus error exception occurs when the MMU detects that a successful address translation is not possible. The MACH vm_fault handler routine is called to do the necessary page lookup and update, etc.

Interrupt Handling

When a peripheral device requires the services of the CPU or is ready to send information that the processor requires, it may signal the processor to take an interrupt exception. The interrupt execution transfers control to a routine that responds appropriately. The device uses the interrupt priority level signals (IPL0, IPL1, IPL2) to signal an interrupt condition to the processor and to specify the priority of that condition.

The status register of MC68020 contains an interrupt priority mask (bits 10-8). The value in the interrupt mask is the highest priority level that the processor ignores. When an interrupt request has a priority higher than the value in the mask, the processor services that interrupt. Priority level 7 is the nonmaskable interrupt (NMI), generally assured when power failure occurs.

This interrupt priority level facility (SPL) is also used in the kernel for synchronization and protection of critical sections from interrupts.

4.4.2 Thread Support

A thread consists of a processor state necessary for independent execution. A kernel stack is allocated for each newly created stack. This kernel stack is used for program execution in the kernel mode. In user mode, a separate user stack is provided. When the kernel switches execution among the runnable threads, the kernel stack is also switched and the states of the threads are saved/restored. The processor state of a thread is stored at the bottom of the kernel stack.

The two structures used to describe the various states of a thread are :

struct mc20_kernel_state (thread context) :

> This structure (d2-d7, a2-a7, PC, IPL) corresponds to the kernel registers as saved in a thread context switch.

struct mc20_saved_state:

> This structure corresponds to the state of user registers as saved upon kernel entry(by traps/interrupts). This is stored in PCB (processor control block) structure in the kernel per-thread structure. It is also pushed onto the stack for exceptions into the kernel.

Save_context() routine saves the thread context and load_context() routine restores the thread context from the base of kernel stack of caller thread. Switch_task_context() routine is used to switch from the currently running thread to a new thread. If both threads belong to the same task, this switching involves only the thread context switching (save/restore the thread state from the kernel stacks). Otherwise, the MMU hardware is updated to refer to the proper page tables (section 4.5).

A new thread is created by a call to thread_create(). The machine-dependent sequence for this new thread bootstrapping can be summarized as :

1. Set up stack & PCB as if the caller had trapped from user space.

2. A dummy frame0 type Exception Frame is created in the stack.

3. return from the kernel as if returning from an exception.

4.5 Porting Virtual Memory Related Code

Virtual Memory in MACH has been designed in such a way that its primitives are simple and general enough to be implemented on any paging system. This is best reflected in the fact that all the machine dependent portions are completely separate from the machine independent parts. MACH VM con- consists of four components: Resident Page Tables to manage physical memory, Address Maps to manage non-overlapping ranges of virtual addresses of tasks, Memory Objects which provide means for external memory management, and Physical Maps ("pmaps") which are machine dependent software structures corresponding to the address maps. For the Horizon III, these correspond to the hardware page tables. Porting MACH VM therefore primarily involved porting the Pmap Management module which gives a machine independent interface to the rest of the VM code.

Before MACH VM could be integrated with the rest of the kernel, we had to:

1. Develop a specialized MMU driver to facilitate the implementation of the Pmaps.

2. Modify resident page table management routines to take care of paging of page table pages.

3. Implement routines to manage page tables referred to by the MMU during the context switch of threads.

4. Implement the VM initialization code which is executed on kernel startup.

5. Write the VM fault handler executed from the low-level trap module ("locore") to handle validation and protection violations in both kernel and user modes. Special attention was given to recovery from vm-faults in the kernel mode.

6 Write various miscellaneous routines required to

integrate VM and IPC (e.g., copyinmsg(), copyoutmsg() among others).

Though the modular implementation of MACH VM helped us a lot, we had to take special care to maintain the hardware consistency of virtual to physical mappings whenever they were updated or deleted, especially during the context switch of threads not belonging to the same task, and during switching of the processor state from the kernel mode to the user mode and vice-versa.

4.6 Kernel Startup

Here we describe how the PAWAN kernel comes up. First, the kernel is bootstrapped enough to run with virtual memory since at the time the PROM loader loads it onto the physical memory, virtual memory is not up and only physical addresses can be accessed. Bootstrapping involves creating a fixed number of kernel page tables at the end of the text and data regions in the physical memory and mapping virtual memory corresponding to the whole of the physical memory until some maximum virtual address. This completes the machine dependent part of VM initialization.

The resident memory module which manages the available physical memory (starting from the end of the kernel page table pages and going upto 4MB) is initialized at this point. Once the address map module and the memory object (pager) module are up, the kernel is ready to manage VM.

Interrupt Vectors are then loaded and devices are probed and started up. This is followed by the initialization of task and thread management portions of the kernel after which the timer is started to enable scheduling. The rest of the kernel including IPC and network management is now initialized in a machine independent fashion. Finally, control is transferred to the first user program, "startup".

4.7 Porting Device Drivers

MACH device drivers are similar to their UNIX counterparts. In fact some of the device drivers for PAWAN have been picked up from IITKIX, an experimental thread based operating system[Das89a]. Rest have been coded afresh. Though there are changes in the way information regarding device addresses and Q-bus structures are allocated, the functionality remains the same. Another addition has been that of asynchronous input interface to the network and tty drivers.

4.8 PAWAN Development Environment

The PAWAN development environment consisted of a collection of SUN-3/60 workstations running SUNOS, connected by 10 Megabit ethernet. The target machine was HCL Horizon III which is a MC68020 based minicomputer. The decision to use Sun systems was taken because they had ample disk space, excellent windowing systems (sun windows and x-windows) and more processing power.

The kernel and all other user level programs were compiled on Sun systems and the object code was transferred to the Horizon machine using "rcp" and "ftp" services. Such an arrangement was feasible because the object code produced on Sun machines could be run on Horizon. This scheme speeded up the development process by a significant amount due to the parallelism it provided -- any member of the group could debug on the Horizon (by booting it) without precluding editing or compilation by others.

GNU software was used throughout for language translation and program management as MACH source code required such support. Revision Control System (RCS) available on the Sun was used to manage the huge amount of source code of our system.

CHAPTER 5

USER SPACE IMPLEMENTATION OF SIGNALS

5.1 Introduction

Signals are used to denote abnormal conditions arising during the execution of a program. Signals are extensively used in UNIX programs and they have been implemented in PAWAN also since one of its design goals was to reuse the existing source code. This chapter describes the design and implementation of signals in PAWAN.

It may be mentioned here that the emphasis of the implementation was on providing the functionality and not the syntactic or even the semantic equality with BSD4.3 signals. The semantics have been redefined to suit the implementation if so dictated by other more important considerations.

The layout of the chapter is as follows -- section 5.2 is an overview of the signals in UNIX followed by a description of some relevant aspects of their implementations. Next we discuss why signals per se are not required in a MACH based system but are needed only for unix programs. Sections 5.5 through 5.7 detail the way signals are provided in PAWAN. Section 5.8 describes how unix style authentication is carried out. Finally there is a general discussion of our implementation and semantic deviations from BSD4.3.

3.2 Signals in Unix

Signals are designed to be software equivalents of hardware interrupts or traps. The analogy can be carried as far as executing a special handler on occurrence of one and masking signals selectively. Over the years the applicability of signals has widened and they are also used for synchronization, asynchronous event notification, controlling process state (e.g. stop, start and kill) in addition to the exception handling.

All signals in UNIX have the same priority. BSD4.3 also allows signals to be masked. Basic interface to signal facilities in BSD4.3 is through kill() and **sigvec()**. Kill() is used to send a signal to specified process and **sigvec()** to specify the action to be taken on arrival of a signal. Action could be one of default, ignore or catch. The last specifies a function which is called when the signal is delivered. Default action can be one of :

- Ignoring the signal
- Terminating the process
- Terminating the process after generating corefile
- Stopping the process
- LOCUS added sigmigrate which by default migrates the receiving process.[Wal83a]

BSD4.3 signal facilities were redesigned to remove the defects observed in earlier systems. The important additions were the ability to mask the signals, restarting of interrupted system calls whenever feasible, ability to specify a signal stack and prevention of recursive invocation of signal handler by masking the signal being handled. Also, signal handler is not reset on delivery of a signal. BSD4.3 signals have also been adopted by POSIX except for omission of signal stack and addition of **sigpending()**, a system call finding pending signals.

Signals, though elegant, have been the achilles' heel of UNIX. Initially they were designed to model exceptions and not to function as an IPC mechanism. In fact a private communication with Dennis Ritchie, referred to in[Bac86a] mentions that signals were not meant to be caught, they were designed as events which were fatal or ignored.

3.3 Implementation of Signals in UNIX

UNIX signaling is split into two parts -- posting a signal and delivering it. Signals are always processed in the receiving process's context. When some signal forces the process to stop, action is sometimes performed when the signal is posted. A signal can be posted by another process (using **kill()**) or by code executing at interrupt level using **psignal()**.

Posting a signal principally consists of adding the signal to a process's set of pending signals. Process is also set to run unless sleeping at non interruptible level (section 5.9.1. The signal posting routine (**psignal()**) also carries out some implicit actions such as stopping the process if it is sleeping and signal action will stop it.

Signals are detected when process returns from sleep, in asynchronous system trap (ast) handler i.e. when cpu mode changes from system to user, or when process returns from a system call or trap. In each of the above cases, a call to **issig()** is made, which checks if a signal is pending, if so it arranges to invoke the handler or take default action. The arrangement consists of pushing an artificial frame on user stack which calls **sigtramp()**, a special piece of code residing in u-area or at the base of user stack. **sigtramp()** in turn calls the handler with appropriate parameters and resumes the user program state with a call to **sigreturn()** after handler has finished.

It may be noted here that since UNIX has only a single flow of control in one process, signal mechanism is complicated by the fact that execution of the handler should be transparent to the user program i.e. user program has to be interrupted and later restored by artificial means.

5.4 Why MACH does not need signals!

IPC interface of MACH is very convenient for most of the functions signals perform in UNIX. Furthermore it is a multi-threaded system so there is no need to complicate the asynchronous event notification by stopping the user thread and invoking the handler -- a separate thread can wait for the event to occur. Similarly blocking/non-blocking send and receive operations render signals unnecessary for synchronization. Finally kernel supports exception handling (in collaboration with exception server) by sending messages on exception port on a per task or per thread basis.

Hence signals are only needed for UNIX programs which use them for non-trivial purposes. Most pertinent examples being shells, which employ signals extensively in job control.

5.5 Signals in PAWAN

The design of signaling mechanism in PAWAN was motivated by the fact that either programs do not use signals at all or they require a reliable implementation. Another motivation was that even a user level implementation should not allow runaway tasks to defy SIGKILL. The latter necessitates some form of kernel support viz. having access to a task's kernel port.

The key idea is to have a signal thread in each task and implement signals by message passing. That is --

signals are posted in the form of a message to a per task[1] signal port and are delivered by a per task signal thread which waits for messages on this port. Since the task sending the signal need not have rights to child's signal port, some other naming mechanism is needed. A signal server is also incorporated to serve as a central agent for deciphering the name and redirecting signal request. The schematic is drawn in figure 5.2.

5.6 Signal Users

Almost all of the signal state is maintained by the user program itself. The rationale being the fact that user can change it at will (using system calls) even if it is maintained the UNIX way, except for signals which can not be caught or masked. These special cases are handled by the signal server (henceforth referred to as SS, section 5.7).

The signal related variables which are stored in u-area and proc structure in BSD4.3, are all stored in the user address space at fixed locations in PAWAN. This simplifies their handling at the time of execve() (section 6.7) and more importantly, minimizes the IPC overhead between user and SS.

[1] In UNIX programs there will be only one main thread so the question of signaling on a per thread basis does not arise

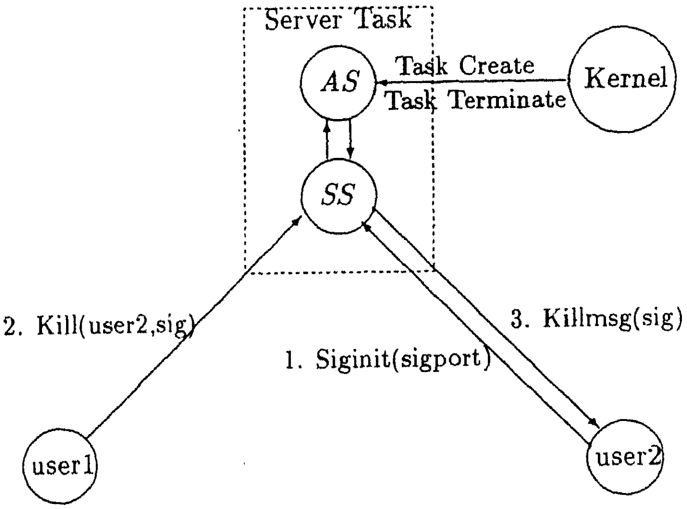

Figure 5.2: Sending a Signal

To clarify the interactions between the user and the SS, the sequence of actions at the time of **sigvec()** is illustrated in figure 5.2. State initialization is done in first call to **sigvec()** when it creates a signal port and passes the send rights to SS, and also creates a signal thread which listens for the messages on the signal port. Signal state initialization is separated from program initialization to avoid extra overhead in tasks which do not want to use the signal facilities. Rest of the **sigvec()** does some sanity checks and installs the signal handler.

When a signal is to be sent to some other process using **kill()** call, the arguments are sent to SS. The victim process is identified by the means of its pid (section 5.8). SS does table lookup to find the signal port of the victim. A non-existent signal port implies that the victim task does not intend to handle the signal, and hence the default action is presumed. Default action is performed on the kernel port of the victim task.

Since all the calls pass through SS, it can take care of the non catchable and non maskable signals -- performing the default action by itself. This guarantees that a mechanism exists for killing or stopping runaway tasks. There can be a slight anomaly here from the user's perspective -- user can effectively change its state to supposedly catch a non catchable signal e.g. SIGKILL (by

```
sigvec(signum, sv, oldsv)
        struct sigvec *sv;
        struct sigvec *osv;
{
        if (bad_sig(signum))
                return ERROR;
                        /* signum is non catchable */
        if (first_sigvec_ever) {
                unix_thread = thread_self();
                sigport = port_allocate();
                sigthread = newthread_crete();
                thread_get_state(sigthread, &STATE);
                STATE.USP = allocate_sigthread_stack();
                                /* bottom of stack */
                STATE.PC = (int) receive_loop;
                thread_set_state(sigthread, STATE);
                inform_server(task_self(), sigport);
                user_lock();
                thread_resume(sigthread);
        } else
                user_lock();
        /* we hold the uesr_lock now */
        if (ovec != NULL)
                *ovec = get_old_handler(signum);
        if (vec != NULL) {
                if (signum == SIGCONT &&
                        vec->sv_handler == SIG_IGN)
                    {user_unlock();return ERROR;}
                                /* can't ignore SIGCONT */
                else if (maskable(signum) == FALSE)
                    {user_unlock();return ERROR;}
                                /* signum non-maskable */
                install handler;
                install associated mask;
                check if syscall abort requested;
                update state (sig_ignored etc.);
                user_unlock();
                return SUCCESS;
        }
}
```

Figure 5.2: Pseudocode of Algorigthm for Sigvec()

modifying the local mask), but it will still be unable to catch the signal since it is implemented by SS. So the maintenance of the signal related variables is better left to the library routines only.

On receipt of a signal message, the signal thread stops the main thread (to give the semblance of single thread of control and to prevent concurrent execution of the handler and the main thread), consults the signal table and takes either the default action or makes a function call to the handler. The main thread is assumed to be the once which made the first call to **sigvec()**. It is expected that the signals will be used only by the traditional unix programs and recoded programs will not use them -- hence it is a safe assumption to make that there will only be one main thread.

5.7 The Signal Server (SS)

In this section we first describe the working of SS very briefly, followed by some pertinent implementation details.

5.7.1 SS -- Description

Signal server exists as a trusted agent in the signaling mechanism, coordinating the activities of users. Most of its work relates to call forwarding -- receiving kill() request and passing them on after due verification. The main loop of signal server is shown in figure 5.3.

SS exports the following functions to its clientele --

kill()

Send a signal

set()

Set the signal port for the task

```
sig_run()
{
    loop_infinite {
        receive(&request_header);
        dispatch(&request_header, &reply_header);
        send(&reply_header);
    }
}

dispatch(request, reply)
{
    function_code = request->msgh_id;
    switch(function_code) {
    case KILL :
                call _Xkill(req->arg1, req->arg2,
                        ..., &res1, &res2, ...);
                rep->result1 = res1;
                rep->result2 = res2;
                .
                .
                .
                break;
    case FUNCTION1 :
                .
                .
                .
                break
                ...
    default :
                rep->return_code = ERROR;
                break;
    }
    return;
}
```

Figure 5.3: Main Loop of Signal Server

reset()

Reset the signal port for the task to NULL.

The meanings of these functions are clear from their names. The last request is used by execve() (section 6.7) since it resets all the signal handlers and deallocates the signal port because there is no advance information that the exec'd task will handle signals.

On receiving a kill request, SS first verifies that sender is authorized to send a signal to the receiver. In case the receiver is not handling the signals (i.e. SS does not know of its signal port) or if the signal is non maskable or non catchable, default action is carried out. Request is passed on in other case. If the request is **set()** or **reset()**, obvious actions are taken.

5.7.2 SS -- Implementation

A brief discussion of the implementation of SS will be in place. In the current version, the data structures used are arrays and searching strategy is linear search, which is more efficient than other popular methods in absolute terms for our small environment. Alternate strategies can be plugged in at a later date as the environment changes.

While the user's idea of the task is its pid, SS deals with a pointer to the entry corresponding to that task in its tables. The type translation is achieved by specifying intran and outtran options in the MIG definition file. (section 3.2.1)

As discussed in section 5.8, Authentication Server (referred to as AS from now on), though logically independent, is structured as a thread in the same task as SS for efficiency sake. This creates race problems and simple locking is likely to deteriorate the performance. To obviate this threat, two level locking[Acc86a] is used in

```
siglock()
{
loop:
        busy_wait(lockl);
        if (lock_try(siglock) == FALSE) {
                unlock(lockl);
                give_processor_to_other_thread();
                goto loop;
        }
        /* we have locked siglock now */
        unlock(lockl);
        return 0;
}

sigunlock()
{
        busy_wait(lockl);
        unlock(siglock);
        unlock(lockl);
        give_processor_to_other_thread();
        return 0;
}
```

Figure 5.4: Algorithm used for locking

conjunction with hand-off scheduling[BLA90a] to access the shared data structures. The siglock is required to manipulate the table while lockl is required to access the siglock itself. First lockl is acquired by busy waiting. This is acceptable because the time for which lockl is held is low (figure 5.4). Once lockl is acquired, siglock is tried. If successful, the thread goes ahead. If unsuccessful, it knows that the other thread should be holding the siglock so it hands-off the cpu to it using the **thread_switch()**. Unlocking requires acquisition of lockl before resetting siglock. It then hands to CPU off to the other thread since it might be waiting for siglock.

5.8 Authentication and Pid's

MACH does not have global task identifiers -- each task has a local name space in which other tasks exist as port names. It must be stressed that for MACH applications it is possible to maintain security in spite of this since ports are kernel protected capabilities. In such applications the temporal existence of a port is used as the unique-id for associated object i.e. so long as the port exists, it is guaranteed to refer to the same object but once it is deallocated, the same name may denote a different object.

Unix-style pids are required solely for unix applications. Kernel port can not be used in its place because it implies more than pids do, viz. the control access to some task. These, coupled with some more information on authentication are maintained by the AS. Since SS is a heavy user of this information, the AS exists as a thread in the same task as SS to minimize the context switching overhead.

AS is limited in functionality and is not supposed to implement a general purpose authentication mechanism. The idea is to have servers implement their own security policies independently. Relying upon a central mechanism makes their credibility dependent on it.

In PAWAN, AS assumes small kernel support in the form of messages on task creation and termination so that it can maintain up-to-date information. It maps the kernel port of the task to its credentials, initializing them from those of the parent. Interestingly no pid allocation algorithms is implemented; instead the names of kernel ports of tasks in its local name space are used as pids -- effectively the unique identifier allocation algorithm which kernel uses for port name allocation is made use of.

Credentials of a task are initialized from those of the parent, but a new set of credentials can also be installed for a task. Such an update is allowed only to privileged tasks viz. those with super user credentials. Functions like **setuid()** can be implemented using this.

Exec Server requires more than changing credentials. In UNIX pid is the parent's notion of child. So even if child exec's a setid file, its pid should be maintained, something which is not possible if a setid file is exec'd in a new task (section 6.7.2). To circumvent this problem AS exports a function **givepid()** to privileged tasks. It creates a new task with the given pid and hands it back to the caller.

Other useful calls provided by AS are **dump_by_pid()** and **pid_to_kport()**, the former produces a unix ps kind of listing and the later gives access rights to kernel port of argument task to privileged programs. These can be used with kill() to implement a supervisor program to kill runaway tasks.

5.9 Discussion

5.9.1 Semantic Deviations from BSD4.3

The implementation discussed in the preceding sections differs with BSD4.3 signals in many ways. Some of the important ones are discussed below.

Treatment of Handlers on exec()

PAWAN deallocates the signal thread and port and informs SS likewise. In BSD4.3 caught signals are reset to default while ignored signals remain ignored. There seems to be no particular advantage of this since a neophyte program can not make any assumptions about its signal state unless it is customized to do so -- which is an extremely unlikely case.

Achieving this semantics in PAWAN is not easy in low cost because the information about the ignored signals is kept in the user's space and is interpreted by the signal thread. If it is required that ignored signals remain ignored across **execve()** either a signal thread will be required to interpret the message or information will have to be maintained by SS.Since there are no a priori means to ascertain which tasks will require signal thread, one thread will have to be provided in each task. The other alternative will mean manyfold increase in communication with SS. Overhead of none of these is justifiable.

Signal Stack and System Call Interruption

The sigstack() call in BSD4.3 allows the signals to be taken on a separate stack, the default being the user stack. The intention behind it being that applications running on a stack which is not automatically expanded should be able to handle signals properly, and probably also that such a facility will carry the hardware interrupts versus signals analogy still farther. But this mechanism has its own drawbacks. (consider failure of signal mechanism on occurrence of SIGSEGV i.e. segmentation violation, on too small a signal stack.)

Since signals in PAWAN involve a separate thread, all the signals are delivered on a separate stack by default. It is possible to deliver signals on the main stack but this will be accompanied by a system call abort (**thread_abort()**,[Bar88a]). This is so because it not possible to restart the system calls at user level.

Delivery of Signals and Sleeping Tasks

BSD4.3 has a notion of interruptible and non interruptible sleep. In our case such an idea does not exist -- in fact the main thread may be sleeping inside kernel when the signal handler executes. We contend that this does not hamper the functionality since the signals are asynchronous events and no assumption about the time of occurrence of signal can be made in general.

Stopping Tasks

Consider the following scenario --

1. A task (pid p) catches, say, SIGUSR1 but not SIGTTIN.

2. SS receives request kill(p, SIGTTIN) and passes it on.

3. Signal thread stops the unix-thread. (default action)

4. SS receives request kill(p, SIGSTOP) and stops p.

It looks as if the task is stopped twice, which is not the case with BSD4.3. This is taken care of by resuming the task on SIGCONT as well as passing it on so the signal thread can resume the unix-thread also.

5.9.2 Conclusion

It seems that the signals, as implemented in PAWAN, are adequate for most of the programs. Due to lack of time, comprehensive testing could not be carried out so data regarding the efficacy of this mechanism is not available. In view of the fact that MACH programs will use more suitable schemes for signaling purposes and signals are needed only for unix programs (section 5.4), compromises on minor semantic details are acceptable.

CHAPTER 6

Execution of a Program

6.1 Introduction

This chapter discusses the design and implementation of **execve()** and Exec Server for PAWAN in the background of corresponding facilities in various variants of UNIX. Section 6.2 discusses the notion of a process, followed by a brief description of process creation and new program execution in UNIX systems. Section 6.4 is a critical evaluation of the such implementations. Following sections discuss PAWAN implementation in detail. **Wait()** is outlined thereafter. The chapter ends by describing shortcomings and proposed enhancements to the current state of the program execution framework in PAWAN.

6.2 Processes

A process carries out the Computation, something for which all the paraphernalia of the hardware is assembled. A computation is defined by code which governs steps of computation and data which gets transformed during it. Furthermore, a stack is implicit in a computation which is defined by means of a procedural language. Stack serves other purposes also, an important one being that of scratch pad for automatic variables. These attributes, viewed from the perspective of a Von--Neumann machine appear as code, data and stack segments residing in memory and CPU executing the code.

The description above covers only the macro state of a process. At a micro level i.e. the operating system level, many other entities have to be included in process attributes due to pragmatic considerations e.g. access privileges. Also, processes state follows state transitions of the finite state automaton defined by operating system e.g. process state changes from runnable to running when it is scheduled.

The operation of executing a new program is carried out by creating a skeletal process and embedding the required state into it. This basically means allocating and initializing operating system state and other system resources, for example memory, loading the new program in memory and making it runnable.

6.3 Process Creation in Unix

It is achieved by means of **fork()**, which replicates the calling process. Then **execve()** is called to overlay the current image by a new executable program. Brief description of semantics and implementation of these calls in BSD4.3 follows.

6.3.1 Fork()

Fork() creates a replica of the calling process, known as child process. This also entails an obvious tree structure on the processes. The child inherits its context from the parent, which includes the state contained in the following ---

* address space

* u-area

* proc structure

File descriptors, signal handlers etc. are duplicated as a result of u-area duplication, while register state, signal state (pending signals, sigmask etc.) come from proc structure. **Fork()** appears to return twice --- in parent it returns the pid (process id) of the child, while in child it returns 0.

6.3.2 Execve()

In BSD4.3 **execve()** is provided to load and execute a new The new program could either be a machine executable or a script to be interpreted by a shell. For an interpreted script, shell is executed instead and the script filename is passed to it as argument. File Formats

Machine executable files in BSD4.3 have a header viz. exec--header prepended to them. This header describes sizes of various segments, loader format, symbol table address and relocation information. The magic number field of header determines the loader format which could be one of the following ---

OMAGIC

Combined text and data segments with writable text

NMAGIC

Distinct, page aligned text and data segments with read only, sharable text

ZMAGIC

Page aligned text and data segments, with sizes multiples of page size. File is demand paged, read only.

Semantics of **execve()**

Arguments to Execve() are: name of the new program, argument vector and environment variables. It never returns if successful since the image which called it is lost. The program specified as first argument is run in its place with given argument vector and environment variables. But all is not lost --- open file descriptors remain open across **execve()**, unless they are explicitly marked close on exec by an **fcntl()**. The exec'd program sees the old descriptors, which might have been changed after **fork()**. This is the way I/O redirection is achieved.

If any of the setid bits are asserted in the mode of the exec'd program, new process is run with the privileges of uid/gid associated with the file, as decided by the mode. Also, caught signals are reset to default while ignored remain ignored. This issue has been discussed in section

5.9.1. The new process runs independently of parent except when it is traced.

6.4 Critical Evaluation of Previous Implementations

Experience with UNIX has helped evolve certain alternate strategies which circumvent the factors producing inefficiency and undesirable semantic effects in the above implementations.

A common observation is that **fork()** is generally followed by **execve()** and the amount of code between these is very small --- chiefly pertaining to I/O redirection and piping. The cost of duplicating the entire address space of parent in the child is wasteful in itself, throwing it away at the time of **execve()** is costly too, if it has to be discarded shortly. Moreover, if the parts of parent's address space are on secondary storage, they have to be brought back im main memory, which in turn triggers excessive paging further deteriorating performance. To reduce the cost of **fork()** different schemes have been used in different variants of UNIX.

UNIX System V duplicates address space by means of copy on write mechanism which defers physical copying of a page until it is modified[Bac86a]. This scheme, however, still entails allocation and initialization of page maps. This operation itself is redundant if the pages are not going to be accessed.

Another idea is to do away with the notion of **fork()** altogether and have alternate means to create processes with known requirements. This idea is used in MACH. With the parent's ability to perform kernel operations on behalf of child, means can be devised to perform I/O redirection, which is the most important factor necessitating **fork()**. Also, parent can read/write child's address space and can mark specific regions of address space for read-write sharing or copy-on-write sharing with the child. This way desired functionality can be achieved without incurring any overheads.

Once parent has the ability to write child's address space, another variation can be sought. In traditional UNIX systems, loading is done inside kernel in course of **execve()**. This restricts the choice of object file format and lessens the scope of experimentation with new loading schemes. Also the load-point (USERTEXT) is hardwired in the kernel thus making it impossible to test/debug programs with different load points targeted for different systems or PROM monitors. OSF recognized this situation and has implemented another system call **exec_with_loader()** in its OSF/1, which loads a user space loader and expects it to load the program. It also has a loader switch table and scans it to find out which loader recognizes the given object file. Thus new object file formats can be conveniently added to the system without necessitating excessive modifications.

6.5 The PAWAN Approach

In view of the above discussion, PAWAN implements **execve()** and **run()** with I/O redirection. The design goals were the following ---

* UNIX semantics should be followed as closely as possible to facilitate reuse of existing source code.

* Execution of setid files should not lead to breach of security.

* Kernel should not be modified to accommodate these calls.

While less important semantic details can be ignored, file table duplication is necessary for execution of almost any UNIX program. Also, BSD4.3 defines that ignored signals remain ignored and caught signals are reset on **execve()**. There seems to be no particular advantage of this and in PAWAN implementation, all signals are reset for efficiency sake.

Fork() is cumbersome to implement in MACH and has been dropped in favor of **run()**. The programs which do not call **execve()** after **fork()**, are the ones which want to run multiple instances of themselves, typically servers and multiprocessor simulators. Since iitkmach is multithreaded, **thread_fork()** (provided in library) is better suited for such applications.

Execve() could be performed by either a server or the calling task itself. A server based implementation is simpler but it incurs extra IPC overhead. Hence it is desirable that exec'ing task performs all the operations by itself. A server also exists for taking care of special cases.

6.6 Useful MACH System Calls

This section quickly reviews the mach system calls which have been used in implementing **execve()** and related calls. Detailed accounts can be found in [Bar88a]. Following sections describe the design in detail and also how these system calls have been made use of.

The MACH philosophy of performing kernel operation on a task by sending messages on its kernel port is seminal for implementing **execve()** and Exec Server, the ES. In PAWAN implementation the flexibility in controlling virtual memory of a task haave been extensively used. Important calls and associated features are described below ---

Virtual Memory Operations

Vm_region() returns a description of the specified region of the target task's virtual address space. System calls **vm_allocate()**, **vm_deallocate()**, **vm_read()**, **vm_write()** support controlling the memory map of the target task. Also, MACH supports a sparse address space, a potential use of which is to have red zones, e.g. to prevent one thread's stack overflowing into another's. Another facility is to force space to be allocated in desired range.

Inheritance of Memory

A parent process may specify that pages of its address space be inherited by the child task in one of the three ways --- shared, copied and absent. This mechanism is useful for implementing a unix-style **fork()**, and also for copying selected regions into child's space thus minimizing overhead.

Vm_map()

Virtual memory mapping is a very general facility which allows a task to specify an agent for handling page faults occurring in given range of its memory. This interface is used for demand paging by specifying the file_pager as the fault handler for the regions where program is to be loaded. An intelligent file pager can also implement text--sharing.

Task and Thread control

Task_create() and thread_create() and system calls create an empty task and a flow of control mechanism respectively. The execution state of the thread can be altered using thread_set/get_state() calls.

Reqestinq Notifications

Mach_port_request_notification() allows a request for certain port related event to be posted. Request could be either of NOTIFY_PORT_DEAD_NAME, NOTIFY_PORT_DESTROYED or NOTIFY_NO_SENDERS. Request specifies two ports --- an argument port whose status change should be notified, and the notify port on which notification is expected.

6.7 Execve() in PAWAN

The pseudocode of the **execve()** algorithm is given in figure 6.1. In order to load the file, **execve()** must be able to write the code and data segments in the memory. This will not be possible if the code of **execve()** itself resides in the code segment hence linking the file loading part of **execve()** with user program is infeasible. The solution that we have adopted is to have the code of **execve()** (henceforth referred to as exec_th[1] reside in a fixed region of memory, its stack lying adjacent to it. As is shown in figure 6.2, this region is below the load point of user programs viz. USERTEXT so exec_th can safely manipulate all the program memory. Exec_th is artificially loaded in the startup task and passed down the task tree. by inheritance when **run()** is called to

[1] exec_th is actually a misnomer since there is no separate exec thread. The code of execve() is executed in the context of the thread which calls it. The name exec_th is carried over from an earlier design in which a per task thread was created to perform exec.

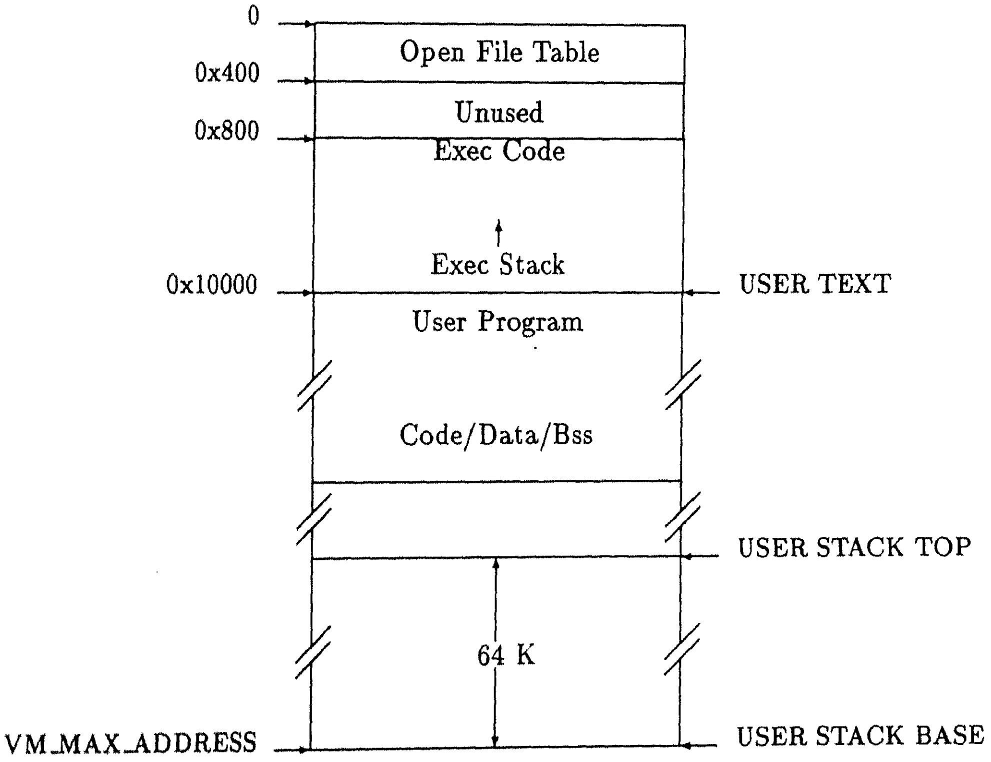

Figure 6.2: Memory Map of a User Process

```
execve(path, argv, envp)
{
        fd = open_file(path);
        check_file_attributes(fd, &is_set_id);
        if (is_set_id == TRUE) {
                contact_exec_server;
                return EXEC_SERVER_WILL_EXEC;
        }
        header = get_file_header(fd);
        shell_name = get_shell(header);
        if (shell_name != NULL) {
                close(fd);
                fd = open_file(shell_name);
                check_file_attributes(fd);
                header = get_file_header(fd);
                is_set_id = FALSE;
                           /* no more shelling */
        }
        clear_signal_state(task_self());
        /*
         * construct a stack as it should look to
         * user process. Return the beginning and
         * end of this stack image in arg_page_begin
         * and arg_page_end.
         */
        usp = process_stack(argv, envp, &arg_page_begin,
           &arg_page_end);
        kill_all_other_threads(task_self());
        kill_ports_other_than_file_system_ports(task_self());
        push_parameters_on_exec_th_stack(task_self(), ...);
        call_exec_th();
        /* NOTREACHED */
}
```

Figure 6.2: Pseudo-code of Execve() library routine.

create a new task.

When a thread calls execve(), it does as many operations as possible by itself and then passes control to exec_th by a non-local goto. These initial operations consist of checking the mode, executability and accessibility of the program file, preparing the task for execing by killing threads other than the calling thread, deallocating ports and resetting state in various servers.

Argv processing is also done at this time i.e. an initial stack for the user program is set up with supplied argument vector and environment. This step is a bit tricky since the stack is set up in a newly allocated region of memory but the pointers in vectors argv and envp should be valid after relocation of this stack to user_stack_base. The address and size of the memory holding the user stack image, along with file descriptor of the program file and a few more parameters are pushed on the exec_th stack and **setregs()** is called which effects a non-local goto to exec_th

Exec_th (figure 6.3) is a separately compiled piece of code which is loaded at address ex_th_begin. It is coded in a re-entrant way and is designed to run as a co--routine with the user program. With the help of volatile declaration of ansii c, all the variables are forced to be in memory, thus we avoid saving registers when a context switch to the user program is effected and only stack pointer (sp) and frame pointer (fp) need be saved. Return address is pushed on the stack.

Exec_th first deallocates the current user stack and copies the stack which has been constructed for new pro-

```
exec_th(task, argbegin, argend, usp, header, fd)

loop:
    /* free user stack */
    vm_deallocate(task, USTACK_TOP, USTACK_BASE);
    stack_size = argend - argbegin;
    vm_allocate(task, USTACK_TOP, USTACK_SIZE, HERE);
    /* copy the new stack image at the base */
    vm_copy(task, argbegin, stack_size, USTACK_BASE - size);
    vm_deallocate(task, USERTEXT, USTACK_TOP - USERTEXT);
    load_file(fd);
    call_user(usp, header.a_entry);
    goto loop;
    /* NOTREACHED */
```

Figure 6.3: Residual Functions Performed by Exec_th.

gram at the base of user stack area. User stack deallocation is possible because exec_th has its own stack to run on. After this step, memory from USERTEXT to USTACK_TOP is deallocated and program is loaded starting from USERTEXT. All said and done, function set_regs is called to switch to user program. It saves the exec_th sp and fp in known locations and jumps to user program.

Some important special cases are discussed below:

6.7.1 File Table

As discussed in Section 6.5, existence of file table is important for UNIX programs. Since file system returns a port right on open() and does not implement any per task file table. it has to be maintained by the user program itself. As can be seen in figure 6.2 this per task table resides in low memory (starting at address OFT_START). Descriptors used with library calls are indices in this table of ports. Since one memory page (4K in our case) is allocated for the table and it does not use all of it, extra space is used for storing variables which are accessed by exec_th e.g. its fp and sp.

6.7.2 Exec'ing Setid Files

As noted in section 6.3.2, task running a setid program has privileges associated with the file rather than those of the parent. The update of privilege should be allowed only to a trusted agent and not to user programs. In PAWAN an Exec Server (referred to as ES from now on) is run at startup time with root privileges. The execve() library routine detects the need to contact ES in file mode checking phase and does likewise.

There is another dimension to privilege update. Since parent can read/write address space of child, it is possible for a malicious parent to overlay the code in child's VM after privileges have been updated. This is an obvious breach of security. OSF/1 disallows exec'ing setid files if anybody else has access to the task's kernel port. We use an alternative strategy which is more flexible. The child task which is passed to ES is terminated and the prooram is executed by a newly created task.

An implication of it is that the child which is runnina a setid prooram seems to have died from parent's perspective. How this anomaly is obviated is discussed in a section 6.8.

6.7.3 Demand Paged Loading

It is achieved by using the pager interface to the file system using **vm_map()**. The file system implements a paging object and kernel redirects faults occurring in the specified memory range by calling **memory_object_data_request()**, to which pager responds by a **memory_object_data_provided()**. The mechanism behind the movement of data is copy on write which makes it highly efficient.

6.7.4 Server State Update

As attributes of the task change or the new task is born, the state contained in various servers has to be updated to reflect these changes. At present the state update is carried out in following servers:

Signal Server

deallocation of signal port, resetting all handlers to default

File Server

resetting authentication state and incrementing reference count of open files

Authentication Server

Duplicating parent's credentials or modifying them in case of setid file execution. Also resetting pid in later case.

Tty Server

Providing child with a control tty.

Once **execve()** is available, **run()** can be trivially implemented. In its PAWAN implementation child task is passed as an argument. It shares the file mode checking, argv processing and server state update phases of **execve()**. Then it copies the exec_th in child, sets the child state such that it starts executing the exec_th and starts the child. File loading part is taken care of in the child itself.

An important addition to **run()** is the provision to pass a redirection table which has <oldfd, newfd> pairs. The child process's fd_table is altered to reflect these changes before child is resumed, thus achieving redirection.

6.8 Implementation of Wait()

Wait() in PAWAN posts a notification request on the event "death of child's kernel port" (Section 6.6.1).

The notify port is a per task wait port. The request is posted at task create time and **wait()** simply blocks on a **msg_receive()** from this port. In case of setid files, old notification request is removed and send rights to wait port are passed to the ES, which then destroys the old task and creates a new task to run the program (section 6.7.2). ES also associates the wait port passed to it with the newly created task in private data structures and then posts a notification request for this task on it's own. On receiving the notification from kernel it simply forwards it on the recorded wait port. The parent task which was waiting gets a proper notification because the field of mach_msg_header_t data structure (viz. msgh_kind) which is used for sending notifications is not interpreted by the kernel so another task can also send notification messages. (Please refer section 2.3.3 for a description of message passing.)

ES runs a separated wait thread for processing notification requests and the access to private data structures is serialized using locking.

6.9 Current Status and Proposed Enhancements

At the time of writing of this thesis, a primitive version of ES is running and the library calls are provided in libexec.a library. In this concluding section the efficacy of this mechanism is discussed.

6.9.1 Details

The size of exec_th is 37 K. The user program load point USERTEXT is at 10000 hex (64 K). We allocate one page (4K) for open file table at address 0. This also avoids problems which may occur due to return value of malloc() being 0 --- which is a valid address in mach. (In BSD4.3 USERTEXT is at first page boundary and 0 is not a valid address.) The stack of exec_th grows towards low memory from 64 K, which is sufficient for its needs.

6.9.2 Differences from BSD4.3

The principal differences and their ramifications, which have been interspersed throughout the previous sections are summarized below:

* Loader is part of user space.

* Signals handlers are reset on execve().

* Fork() is conspicuously absent.

* Run() has been added.

* Vfork() is missing.

* Implementation is at user level

* Close_on_exec fcntl() is not implemented, it can be incorporated very easily though.

* Ptrace() is not provided since it is trivially easy to do with the MACH system calls using child's kernel port.

6.9.3 Advantages

Some of the significant advantages which the above scheme gains are :

* Flexibility with load point

* Different loaders and object file formats can be implemented

* Avoids unnecessary copying of the address space as in fork()

* User space implementation provides flexibility, especially as more servers are added.

6.9.4 Pitfalls

One of the significant shortcomings of this scheme is that the open file table and exec_th are user writable. If a user program inadvertently overwrites this code --- further file accesses may be disallowed. Also operation of signals and execve() is likely to be hampered and the task will be terminated due to address space violation in normal case.

Another reservation that might be expressed is the overhead of having exec_th in every user task. This is not really so serious a problem as it looks because of copy on write mechanism used in virtual memory copying.

Since the code segment of exec_th is approximately 24 K of which only one physical copy will exist in general.

Inefficiency also results from the fact that exec header is not loaded along with the file in OMAGIC and NMAGIC formats. Hence the parts of file viz. text and data segments which are to be loaded at page boundary in memory appear at a small offset i.e. sizeof(exec header) in the file. This misalignment prevents copy on write sharing to take place and causes file pages to be physically copied.

6.9.3 Proposed Enhancements

Since one of the goals was to utilize existing source code, file table interface had to be retained. This need not be the case with programs written anew. So file table can be dispensed with by using the ports returned by file server directly. Redirection can be implemented using environment manager[Gop92a]. The scheme is that two ports named "stdin" and "stdout" will exist in environment of every task by default and these will be used for I/O. For redirection, there values can be altered by sending a request to environment manager.

Current implementation reads in the files in memory as BSD4.3 does. A better way is to map the file code in user memory using the memory object interface of MACH [Tev87b]. Such a modification will increase efficiency substantially.

If compilers are modified to generate a different exec header significant improvements will be gained. Two improvements which can be immediately recognized are --

Page aligned segments will improve efficiency of loading phase. Currently ZMAGIC format of BSD4.3 generates text and data segments with sizes multiples of page length. It also includes exec header in text segment size. Unfortunately there is no way misalignment due to header can be avoided in other formats.

If exec header contains USERTEXT as a field, programs targeted to be loaded at other locations than a hardwired system wide USERTEXT can also be executed and tested. Such a facility will be immensely useful for writing programs for other systems and for EPROM monitors. If load point is variable, intelligent programs which perform their own address space management will also be benefited. Currently all utilities, e.g. debuggers are hardwired with respect to a load point which render running and testing special purpose programs very difficult.

It may be noted here that MACH has introduced a new file format, known as MACH-O[Tev87a]. It encourages use of shared libraries, which are not available in BSD4.3. Many other useful primitives like sharing of memory and file mapping are also supported.

CHAPTER 7

Conclusion and Extensions

7.1. Epilogue

The work described in this, and companion theses, lays out a platform for future work, analogous to a hardware backplane on which different cards can be plugged in. Our basic aim was to experiment with a system which could survive in a rapidly changing world. As research has brought out, an extensible system with user state servers is the solution. This thesis has discussed design and implementation of two such servers --- exec server and signal server.

As specified earlier the motivation to take up this works was not to provide a system on which people could log in and work as smoothly as ever right in the beginning --- something which is difficult to achieve in such a short span of time as we had. Instead we strived to investigate and pin down the specific issues and tread-offs which govern the characteristics of an operating system in a scalable, multiprocessor and distributed environment. Similarly efficiency was not our immediate concern (none of the servers use elaborate searching schemes). We believe that with a proper design, efficiency issues can be tuned up in later stages of development. To this end, we have been able to meet our goals, which have been listed on next page.

* Isolating hardware dependencies in porting and mapping hardware characteristics of various machines to enable quick porting

* Providing a workbench for experimental distributed system research

* Exploiting machine and operating system features to improve flexibility and to encourage new programming methodologies (e.g. having multi-threaded servers)

* Designing a communication oriented system to exploit parallelism

* Redefining unix semantics to make it suitable for user level implementations in a multiprocessor environment

* Designing a unix-like system which can be transparently extended using outside kernel solutions, yet maintaining unix source code compatibility with minimal modifications

UNIX was designed for an isolated, uniprocessor system and it flagrantly falls short of expectations in different kinds of system architectures which are prevalent today. One facet of our work was to isolate instances where unix methodology is not the best one to pursue. Appropriate solutions, and more importantly principles involved, have been worked out to tackle such intricate issues.

Since users are more used to UNIX and there is a vast amount of source code existing for the same (not to mention the unix associations), providing a unix-like system was one of our goals. As pointed out at various occasions earlier, unix semantics are difficult to achieve in newer systems. In fact .it is even inappropriate or undefined in some cases (e.g. forking a multi-threaded program). Redefining UNIX semantics to suit PAWAN design and simulatneously being benign to the old programs and new users has been a major achievement on our part. A pertinent example is the subtly changed behaviour of signals.

PAWAN, at the time of writing of this thesis, is not a complete system in itself. It provides most rudimentary services, namely a file system, program execution, global name space in the form of environment manager, a usable interface to the network and unix style signals.

A sophisticated user environment does not exist --- programs have to be edited and compiled either on other systems or in unix environment of the same machine. Also, there is no shell.

We have not been able to gather performance statistics due to lack of time. It is our conviction that efficient design and implementation together with accurate o/s primitives will dominate the context switch overhead, thus gaining in performance over BSD4.3. Future theses on PAWAN will have more to say about it.

7.2 Suggested Extensions

The two aspects of the extension needed are --- to provide a user environment and to add tools enabling extensive distributed and multiprocessor system research.

To provide a program development environment, shell, compiler and editor must be ported. This should be easy as fundamental libraries involving file system, signals and network have been provided. Other utilities can also be ported in an incremental manner. A more comprehensive environment will require more servers e.g. pipe server and authentication server.

The other aspect viz. tools for distributed and multiprocessor system research, is more important. There is ample scope for innovation here as these are not going to hinge upon UNIX compatibility. Following extensions can be immediately sought:

1. Providing full range of network services including network monitoring.

2. Implementing higher primitives --- distributed shared memory and net message server.

3. Implementing a processor server to experiment with multiprocessor scheduling.

4. Identifying issues in process migration and load balancing.

5. Incorporating various I/O scheduling strategies in device server.

An entirely different research direction could be that relating to fault tolerance in distributed systems. Network related tools will be of immense use in such an endeavor and issues relating to replication, checkpointing and rollback, network routing etc. will have to be looked into.

The work presented here has been more like laying down a foundation and it is expected that further research relating to more practical and higher level issues will be taken up in years to come.

References

Tev87a.

A. Tevanian Jr and R. Rashid, "Mach: A Basis for Future UNIX Development," Technical Report CMU-CS-87-139, School of Computer Science, Carnegie Mellon University, June 1987.

Acc86a.

M. Accetta, R. Baron, D. Gloub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," Technical Report, School of Computer Science, Carnegie Mellon University, August 1986.

Wal83a.

A. N. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," Proceedings of the Ninth Symposium on Operating Systems Principles, ACM, October 1983.

Tan90a.

Andrew S. Tannenbaum, "Amoeba - A Distributed Operating System for the 1990's," IEEE Computer, 1990.

Che88a.

D. R. Cheriton, "The V Distributed System," Communications of the ACM, vol. 31, no. 3, March 1988.

Das89a.

P. C. Das and G. Barua, "A Threads Facility for IITKIX," CSI Journal of Computer Science and Informatics, vol. 19, 1989.

Gop92a.

B. Gopal, "PAWAN : A MACH Based UNIX System(II)," M. Tech. Thesis, Indian Institute of Technology, Kanpur, April 1992.

Rao92a.

P. V. Rao, "PAWAN : A MACH Based UNIX System(IV)," M. Tech. Thesis, Indian Institute of Technology, Kanpur, April 1992.

Bar92a.

M. Baruah, "PAWAN : A MACH Based UNIX System(I)," M. Tech. Thesis, Indian Institute of Technology, Kanpur, April 1992.

Tev87b.

A. Tevanian Jr, "Architecture Independent Virtual Memory Management for Parallel and Distributed Environments," Ph. D. Thesis, School of Computer Science, Carnegie Mellon University, December 1987.

You87a.

M. Young, A. Tevanian, Jr., R. Rashid, D. Gloub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," Technical Report CMU-CS-87-155, School of Computer Science, Carnegie Mellon University, August 1987.

Tev87c.

A. Tevanian, R. Rashid, D. Gloub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control," Technical Report CMU-CS-87-149, School of Computer Science, Carnegie Mellon University, August 1987.

Ras87a.

R. Rashid, A. Tevanian Jr., M. Young, D. Gloub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," Technical Report CMU-CS-87-140, School of Computer Science, Carnegie Mellon University, July 1987.

Tev87d.

A. Tevanian, R. Rashid, M. W. Young, D. B. Gloub, M. R. Thompson, W. Bolosky, and R. Sanzi, "A Unix Interface for Shared Memory and Memory Mapped Files Under Mach," Technical Report, School of Computer Science, Carnegie Mellon University, July 1987.

Tho88a.

M. R. Thompson, "Mach Environment Manager," Online Mach Documents (unpublished), School of Computer Science, Carnegie Mellon University, July 1988.

Dra89a.

R. P. Draves, M. B. Jones, and M. R. Thompson, "MIG --- The MACH Interface Generator," Online Mach Documents (unpublished), School of Computer Science, Carnegie Mellon University, July 1989.

Coo88a.

E. C. Cooper and R. P. Draves, "C Threads," Technical Report CMU-CS-88-154, School of Computer Science, Carnegie Mellon University, October 1988.

Bir84a.

A. D. Birrel and B. J. Nelson, "Implementation of Remote Procedure Calls," ACM Transactions on Computer systems, vol. 1, February 1984.

Bac86a.

Maurice J. Bach, The Design of UNIX Operating System, Prentice-Hall Inc., Eaglewood Cliffs, 1986.

BLA90a.

David L. BLACK, "Scheduling Support for Cocurrency and Parallelism in MACH Operating System," IEEE Computer, May 1990.

Bar88a.

R.V. Baron, D. Black, W. Bolosky, J. Chew, R. P. Draves, D. B. Gloub, R. F. Rashid, A. Tevanian, Jr., and M. W. Young, "Mach Kernel Interface Manual," Online Mach Documents (unpublished), School of Computer Science, Carnegie Mellon University, October 1988.

Lef89a.

S. J. Leffler, M. J. McKusick, M. J. Karels, and J. S. Quarterman, The Design and Implementation of 4.3BSD UNIX Operating System, Addison-Wesley Publishing Co., May 1989.

Tan89a.

Andrew S. Tannenbaum, Operating Systems - Design and Implementation, Prentice Hall India Pvt. Ltd., New Delhi, October 1989.